Some Ideas on Automatic Theorem Proving

W. W. Bledsoe

May 14, 1973

University of Texas, Austin, Texas

ATP-9

Some Ideas on Automatic Theorem Proving

W. W. Bledsoe

May 14, 1973

The following are some tasks that might be accomplished in conjunction with our automatic theorem proving effort here. Many of these are just ideas and suggestions that need development and testing.

The program referred to here is the man-machine theorem proving program developed at the University of Texas and reported on in "A Man Machine Theorem Proving System", W. W. Bledsoe and Peter Bruell, IJCAI-73, Stanford,California, August 20, 1973.

TASKS

1. Theorem Retrieval
2. REDUCE revision
3. Experience with the program
4. PAIRS generalization
5. Treat PUT f like a definition of f
6. Unknown Predicate signal
7. Substitute equivalence
8. Overlay
9. Parallel Search over-director
10. Untrapping
11. Models

1. <u>Theorem</u> <u>Retrieval</u>.

a. This would anticipate the storage of a large <u>list of theorems</u>(like all the theorems that have ever been published, or all theorems on topology and set theory.)

b. A rapid retrieval mechanism would determine whether a given subgoal is a substitution instance of any theorem on the list.

c. Speed is most important. We want an essentially parallel search mechanism which quickly points to the correct theorem, or quickly shows there is none there. This is no trivial problem, but has a practical solution, I believe. The manner in which the theorems are stored may be of importance here.

d. <u>Display</u>. There should be easy access, a means for the operator to quickly call to the scope a given theorem, displayed in an easily readable form (like "Tree-print").

e. The operator might want to call for all the theorems (in his list) related to certain concepts (predicates) and have them displayed in order on the scope, but under his control so he can "zero-in" on certain ones.

f. The Prover program itself might bring in such a list that is related to the theorem (subgoal) being proved. This could be triggered by a user command "REFERENCE" or triggered by its failure to obtain a proof (I prefer the first).

2. <u>REDUCE</u> <u>Revision</u>.

a. A complete revision of REDUCE seems in order to make it more <u>complete</u>, <u>fast</u>, and <u>visual</u>. Mike Ballantyne has some ideas on this.

b. Speed. No solution to this will be welcomed by me if it is not very fast. REDUCE is a routine that is called in every cycle of the program so it must use very little time.

c. Complete. A systematic effort should be made to include all atomic formulas that should be reduced to something else. This complete list need not be present in our current program but could be prepared for future larger machines.

Mainly though, we need a philosophy on what formulas are to be reduced. Then a program can probably generate most of the entries by

(i) examining the theorem list

(ii) considering certain combinations of predicates and proving the desired theorems, which make up a REDUCE table entry.

For example, the combination $\in \{\ \}$ is always reduced, e.g. $x \in \{t : P(t)\} \Rightarrow P(x)$

Why? Because an elemental set operation "$\in$" is applied to a "higher" (?) set operation? Similarly $\in \cap$. Is there a general principle here that can be applied? How about $\subseteq \cap$?

d. Visual. We need an interactive command for showing the reduce table (SHOW REDUCE) which results in an easy to read display of all or part (triggered by certain predicates) of the table being displayed on the scope under control of the user. (In general, on all such visual displays, we want to be able to STOP the display [by depressing the space bar?] and then continue it.)

e. Add and Delete. Easily add and delete entries. Like our present ADD-REDUCE.

f. Try it out on a number of theorems (proofs).

3. Experience with the Prover.

This involves working with the prover on a large number of examples, to determine its strengths (?) and weaknesses, to develop new, needed, interactive commands.

This is a good job for a pure mathematician, especially if he can occasionally be accompanied by a programmer-mathematician.

4. PAIRS Generalization.

a. What is the *real* concept behind the PAIRS procedure? Why is it (or is it?) more efficient than retaining in the hypothesis and additional formula?

For example, the use of PAIRS on the goal

(1) $(\text{Cover } G_0 \rightarrow \text{Cover } F_0)$

to suggest the subgoal

$(\text{Ref } G_0\ F_0)$ $(G_0$ is a refinement of $F_0)$

is equivalent to having the formula

(2) $(\text{Cover } G \wedge \text{Ref } G\ F \rightarrow \text{Cover } F)$

as an additional hypothesis in (1) and backchaining.

The saving seems to be caused by the fact that (2) is not involved *unless* *two* parts of it are *partially* matched. Notice this parallelism in backchaining: instead of matching *all* of one part of (2) we match *some* of two parts of it.

Can this concept be extended to more levels of parallelism?

(4.) b. Non-alike PAIRS.

We need not have the same predicate on both sides of the implication in order to trigger the PAIRS procedure. For example, if we have the goal

$(A_0 \subseteq B_0 \rightarrow X_0 \in B_0)$

"PAIRS" could suggest trying

$(X_0 \in A_0)$

or, using the example of 4a,

$(\text{Ref } G_0 F_0 \rightarrow \text{Cover } F_0)$

could suggest as a subgoal,

$(\text{Cover } G_0)$.

What are the restrictions? Should we process all theorems in this fashion and get a large PAIR table? (If so, then a better way to do it would be to reference the theorem in the theorem-list rather than make duplicate entries in PAIRS table.)

I doubt that non-alike PAIRS should take preference over alike PAIRS.

c. Quick Check. In processing a PAIR subgoal, especially a non-alike case, it is probably wise to use a shallow search rather than a deep one. We already prevent PAIRS from recalling PAIRS on the same predicate. Carrying this further, I would suggest not getting too deep on a PAIRS search. The theorem lable could be used to control the depth.

For example, maybe we should allow only ground-IMPLY to be called by PAIRS, or allow no back-chaining, etc. (see 9)

5. Treat PUT like DEFINITION.

a. When we now use PUT it usually causes a simple formula like

(function f)

to be replaced by a messy looking one like

(function ($\lambda x \in A$ (Choice {........})))

which is hard to read. Why not treat PUT f (...) like an ordinary definition, whereby f is replaced in the theorem by (f) and the value (...) is stored as the definition of (f) (in the same way that the definition of (reg) is stored).

Then the program can (naturally) retain (f) in its displays, PEEK at the definition of (f) when needed, etc.

DEPUT would still try proving hypothesis as before but a neater display would result.

b. This needs to be tried on some examples before it is permanently installed.

6. Unknown Predicate Signal.

When the PROVER returns the message

FAILED.. or FAILED TIMELIMIT..

it could be required to print, as well, any predicate letter in the conclusion which

(i) does not appear in the hypothesis (or definition of terms there)

(ii) has no definition

(or both).

This would help the user see what definitions need to be added and/or hypotheses required.

7. Substitute equivalence.

In addition to replacing equals it would probably be useful to replace equivalence also. For example, try replacing the goal

$$(\alpha \leftrightarrow \beta) \wedge H \rightarrow P(\alpha)$$

by

$$(H \rightarrow P(\beta).)$$

8. Overlay.

As space becomes scarce and our program gets longer, we need to consider ways of overlaying parts of the program. For example, we could overlay everytime the user interacts, or everytime we did some not-often-performed operation.

Whatever is decided upon must be tested on several examples to see if it can be tolerated by the user.

9. Parallel Search over-director.

Concepts like PAIRS, non-alike PAIRS, and the EXLODE levels of HOA are attempts to obtain a parallel search capability. It seems desirable to have an "over-director" program which constantly reviews the whole situation and directs various parts of the program to act, thus continuing certain lines of attack until a certain depth is reached before switching to another line. The theorem label could possibly be a useful tool for this direction.

If the program is halted on a certain part of the proof, that part can be saved (with its theorem label) to possibly be pursued later. If TL1, TL2, ..., TLN, are the theorem labels from all such partially completed tasks, then the over-director can calculate the desired line of attack as a function of the TLi. This is like tree search heuristics found in the literature.

A built-in bias could be used to make the program "want" to continue with its present line of attack a little longer than normal, thereby giving stability to the proof, and more ease of following by the human user. The user should be allowed to easily override the over-director.

10. Untrapping.

Bill Bennett can explain this. This is a procedure, not unlike the concepts of TYPES found in our limits paper, whereby the computer when given a goal

$$\exists x\ (P(x) \wedge Q(x))$$

proposes a substitution for x to satisfy P(x)

and proposes a substitution for x to satisfy Q(x)

and then reconciles the two solutions, always leaving x as a variable to be further restricted later, if needed. This prevents the "trapping" that often occurs when a substitution is given which satisfies P(x) but which does not satisfy Q(x), and this fact is not discovered until a large amount of computing has been expended.

This is the very heart of automatic theorem proving in my estimation, and remains untouched in the literature.

It remains a challenge to design a "proposer" and "reconciler" to carry out these functions. For example, on the subgoal

(1) $\exists G(G \subseteq \text{Open} \wedge \cup G \subseteq B_0)$

the program might propose the substitution

(2) Open/G as a solution to $G \subseteq \text{Open}$,

and

(3) Subsets B_0/G as a solution to $\cup G \subseteq B_0$

In each case the program has chosen the "largest" value for G which will satisfy the given subgoal. The program will then try to combine the two solutions (2) and (3) to get a solution to (1). In this case an easy solution to (1) is gotten by <u>intersecting</u> the two solutions (2) and (3) to get

$$(\text{Open} \cap \text{Subsets } B_0)/G.$$

For this particular kind of example the "Proposer" could send the subgoal

(4) $A \subseteq A_0$

to a subroutine SOLVE$\subseteq$ which would find the <u>largest</u> set for A satisfying (4). SOLVE$\subseteq$ could be further augmented to handle

(5) $f(A) \subseteq A_0$

for certain predetermined f's (such as $\cup$). The "reconciler" here would simply intersect the two answers, as was done in the above example.

Much work has yet to be done to determine the feasibility of such a propose-reconcile untrapping technique.

11. <u>Models</u>.

Bill Henneman can explain this. He has worked out a program which provides a lattice (or a net) of relationship between sets and their closures, interiors, etc., for all sets mentioned in the proof of a theorem. This, in theory, provides an easy look-up of facts about these sets. I believe Bob Anderson has another or similar approach, and others of you are involved with both. (NOTE: This is now handled as a special case of Ballantyne's and Bennett's paper "Some Graphic Methods in Topology Proofs".)